Front.Comput.Sci.14 – This is the author version, the full text is available at https://link.springer.com/
DOI 10.1007/s11704 – 018 – 7217 – 7

RESEARCH ARTICLE

TRAP: Trace Runtime Analysis of Properties

Daian YUE [1,2], Vania JOLOBOFF [1,2], Frédéric MALLET (✉)[3,1]

1 MOE Trustworthy Software International Joint Lab, East China Normal University, Shanghai 200062, China
2 Inria, 35042 Rennes, France
3 Université Cote d'Azur, CNRS, Inria, I3S, 06900 Sophia Antipolis, France

Abstract We present a method and a tool for the verification of causal and temporal properties for embedded systems. We analyze trace streams resulting from the execution of virtual prototypes that combine simulated hardware and embedded software. The main originality lies in the use of logical clocks to abstract away irrelevant information from the trace. We propose a model-based approach that relies on Domain Specific Languages (DSL). A first DSL, called TISL (Trace Item Specification Language), captures the relevant data structures. A second DSL, called STML (Simulation Trace Mapping Language), abstracts the simulation raw data into logical clocks, abstracting simulation data into relevant observation probes and thus reducing the trace streams size. The third DSL, called TPSL, defines a set of behavioral patterns that include widely used temporal properties. This is meant for users who are not familiar with temporal logics. Each pattern is transformed into an automata. All the automata are executed concurrently and each one raises an error if and when the related TPSL property is violated. The contribution is the integration of this pattern-based property specification language into the SimSoC virtual prototyping framework without requiring to recompile all the simulation models when the properties evolve. We illustrate our approach with experiments that show the possibility to use multi-core platforms to parallelize the simulation and verification processes, thus reducing the verification time.

Keywords Runtime verification, trace analysis, property specification, logical clocks, simulation, virtual prototyping.

Received June 24, 2017; accepted January 30, 2018

E-mail: *Frederic.Mallet@unice.fr*

1 Introduction

Electronic devices have become ubiquitous in our everyday life, including consumer electronics, Internet of Things solutions, transportation systems or industrial processes. These devices combine dedicated hardware with specific embedded software. Deciding on what should be hardware and what should be software is called the co-design process and largely depends on the expected performances and the targeted cost. Virtual Prototyping is a common solution to investigate alternative designs and verify that the candidate system satisfies the requirements. A virtual prototype (VP) is a virtual system that emulates the hardware, using some hardware simulation technique, and runs the application software 'as is'. The virtual prototype outputs the same results as the real system. In addition, the system can generate synthetic outputs by observing the system state and its transitions. Overall, the virtual prototype generates a trace, which can be recorded into a trace file.

A system must satisfy requirements, functional or non functional, such as timeliness. The system is often modeled as a high-level abstraction to detect early problems. Even though exhaustive verification techniques such as model checking can address increasingly larger parts of the system on such abstract models, system integration requires complementary techniques that consider the system as a whole to check that the required properties are satisfied. For known required properties, assertion-based verification techniques are widely used [1, 2], typically by running many simulations with varying parameters and varying

interactions, checking that the assertions hold. Assertion techniques work well when (i) the properties are known in advance and (ii) the source code of the virtual prototype is available to introduce assertion checks.

However, most often during the design phase engineers discover some system failures that are not captured by the assertions because not anticipated. Investigating the cause of the failure may not be easy because it may derive from a long chain of events that eventually lead to the failure and there are often some interactions between both new hardware and new software designs, both having their own flaws. One of the traditional ways to find such design errors from VP simulators is to analyze their traces. A trace stream usually consists of the sequence of mixed events, states and transition steps with variable values, possibly annotated with time stamps. By analyzing multiple trace files, engineers may eventually discover the problem by successive iterations, typically searching in the trace stream at which point it deviates from the model. They also often discover new properties that should be verified but were not formulated in the initial requirements and therefore should be added as new assertions. Trace analysis is thus very useful as long as trace analysis tools are convenient enough to be handled by seasoned engineers and powerful enough to discover intricate properties. There are two major problems during this kind of analysis:

1. It is hard to extract useful information from a huge trace file. A trace file generated by a VP simulator often contains very detailed information about the system, including some internal states, and the trace file can reach gigabytes of data, which is hard to handle efficiently.
2. It is difficult to reason about the very detailed traces. Because the trace contains raw binary data and not symbolic information, it may not be obvious to reason about a long chain of causality events that eventually leads to a failure. The low-level detailed trace information makes it hard to reason at an abstract level.

We propose our Trace Runtime Analysis Platform (TRAP), a model-based framework that supports the runtime analysis and verification of traces generated by VP simulators. The framework that we propose aims at achieving several independant objectives combined in the approach

- Offer to engineers an intuitive property specification language. The targeted properties are related to facts that should be observed, or should never occur, possibly with some time constraints. We believe embedded systems engineers should not be required to be familiar with temporal logic or other formal methods. The language we propose only requires standard logic and usual mathematics. The language also includes primitive expressions for common notions in CPS such as throughput or burstiness.
- Make it possible to specify properties that are required only in some phases of the system. For example, a system may have three phases, the boot phase, the running phase and the shutdown phase. Different properties may have to be valid in these phases and the transition between phases must be captured.
- Support reasoning on symbolic data, i.e., we propose a mechanism to map raw binary trace data into symbolic data and ignore irrelevant data. This mechanism makes it possible to reason on abstract properties and reduce the size of trace files. We rely on the notion of logical clock for such an abstraction.
- Combine the above goals within a model driven approach. Our approach is using three separate Domain Specific Languages, to describe the trace data, to abstract trace data into symbolic information, and to express properties using a user-friendly specification language editor that does not require temporal logic background. The Eclipse Modeling Framework is the basis of our implementation.
- Propose a dynamic runtime analysis framework that does not require recompiling the virtual prototype or re-building the virtual prototype. Our framework uses dynamically loaded modules to verify the properties in different simulation sessions without recompiling the virtual prototype, which can be tedious and sometimes just not possible when some source code is not available. With this technique, engineers can experiment various sets of properties with a fast iterative development cycle.
- Provide a fast implementation of property analysis that can be used as runtime verification and does not slow down the performance of the virtual prototype. The implementation is constructed such that the property verification is an independent tool that can run in parallel of the virtual prototype to analyze the trace stream.

Our goal is to make it easy to check for properties by quickly analyzing trace streams with a powerful tool able to verify causal and temporal properties, while reducing the

size of trace streams. Our approach does not require rebuilding the simulator when adding new properties. Raw binary data are abstracted into a formal trace that can be either stored into a file for later usage, or analyzed in real time at run time to detect property violations without slowing down the simulation.

The paper is structured as follows. The next section presents some background work in the trace analysis and property specification areas and the background technology used, in particular the relationship with CCSL. Section 3 presents our global approach, decomposed into several steps, based on a modeling approach and model transformation using Domain Specific Languages (DSL). Section 4 presents the details of the tool function and an example drawn from a System on Chip simulator.

2 Background and Related work

2.1 Related Work

Even though model-driven engineering approaches with model verification techniques are used, the engineers still face the problem of debugging simulation models [3] and runtime verification is still needed in most industrial applications [4]. Trace analysis has been a topic of work for some time in the simulation community. Several languages in the literature allow to express temporal properties, among which Linear Temporal Logics (LTL) [5]. LTL abstracts the system behavior as an infinite sequence of states and establishes either properties of *invariance* or *eventuality*. All the properties are expressed relative to the steps of execution, using modal temporal operators, and not relative to an external (physical or not) clock measure. Several efforts have been made to provide real-time extensions for temporal logics (RT-LTL and others).

Later, some efforts were made to renounce to the full expressiveness of some logic in favor of a representation more natural to the designers and more computationally tractable. This lead to the notion of Logic Of Constraints (LOC) initially proposed by Balarin and al. [6] and used for trace analysis by Chen and al. [7]. This system allows to express functional and performance constraints containing event names, instances of events, index variables allowing to express generic formulas. For example, the formula $t(Stimuli[i+1]) - t(Stimuli[i]) < T$ denotes the property that two successive occurrences of the event $Stimuli$ must occur within a given time frame T.

As SystemC is widely used by the embedded systems community, the specific VCD (Value Change Dump) trace format has been defined for signal tracing and many commercial or non commercial tools exist for tracing wave forms. A more flexible trace generator is proposed by CULT [8], but there is no associated tool to analyze the traces produced. A SystemC analyzer like [9] can be used to verify temporal properties holding in the SystemC processes, but it requires access to the SystemC modules and cannot check information originating from embedded software running over the simulated hardware. Although the experiments related in this article are based on SystemC, nothing in the approach requires SystemC. Our DSL makes it possible to check value changes on any variables, not restricted to signals.

The COSITA tool [10] analyzes traces resulting from co-simulation between SystemC and Matlab models for automotive applications. The tool permits to detect differences between simulation and emulation to investigate models but it does not include a property specification language. The Meta-Event Definition Language (MEDL) [11] is used to verify properties in traces of packets in network simulation, using the Monitoring and Checking (MAC) framework [12]. It is based on a logic for events and conditions, which allows to consider instances of events by means of counters, similar to LOC.

Referring to property specification language, Dwyer *et al* published in 1999 a classic paper [13] which did a survey from many sources and developed a pattern system. Their pattern system covers most popular situations. Based on their research, another paper [14] builds a new pattern system containing some real-time extensions. Also, they studied many formal specification languages and provide a rather convenient English-like grammar. However, their paper does not refer to the implementation, thus there is no information on how to apply their pattern system to the real simulators or to actual simulation traces.

Considering specifically trace analysis, a similar, though different approach, consists in considering usage scenarios and either generate timed automata that can be model-checked such as [15] or generate tests associated with the verification of pre and post conditions [16]. The work from [17] has extended this mechanism to verify some temporal relations based on traces. Yet another approach is to replay the execution of a trace over the model to investigate the example at stake [18].

Regarding the tooling aspect, the Open Trace Format [19] provides a flexible trace format that inspired our trace

mapping work. Our mapping system is using the notion of meta model for defining a trace format introduced by [20].

2.2 Relationship with CCSL

The Clock Constraint Specification Language (CCSL) [21] is devised for capturing chronological, causal and timed relationships. It is based on the notion of logical clock, that was promoted concurrently by Lamport in the synchronization of distributed systems [22] and by Berry et al. at the heart of synchronous languages [23]. A logical clock is a totally ordered set of *ticks*, which can be identified by their *index*. CCSL is different in expressiveness from LTL (see [24]). It provides a set of safety patterns regarding both causal and temporal properties. It combines purely boolean expressions with unbounded counters, thus being more expressive than regular languages and adequate for properties such as targeted here. CCSL defines an algebra of clock expressions and relations. Some operators are bounded, others are unbounded. We provide here a summary of the clock operators that are used in TRAP. In the sequel, we use the word tick to mean an occurrence of a logical clock. The most important expressions and relations of CCSL are listed below, but the reader should refer to [25] for comprehensive and formal definitions.

Union The union of two clocks is the clock that ticks whenever either one of the two clocks ticks, denoted in CCSL $a + b$, where a and b are clocks.

Intersection Denoted $a * b$, it is the clock that ticks each time the two clocks a and b tick simultaneously.

Subclock. Denoted $a \subset b$, subclocking is a relation between two clocks that forbids clock a to tick when clock b does not tick.

Sampling. The sampling clock has two clock parameters, i.e., the trigger clock and the base clock. It ticks in coincidence with the base clock immediately following a tick of the trigger clock. It is denoted as $c = a \setminus b$.

Defer. Denoted $a(n) \rightsquigarrow b\ ns$, a deferred clock has two clock parameters, a trigger clock (a) and a delay clock (b), and an integer parameter (n). Every tick of the trigger clock starts up a counter. For every tick of the delay clock, values of the active counters are decreased. When the value of counter reaches 1, the defer clock occurs. In TRAP, defer clock to infinity are invalid.

Precedence. Denoted $a \prec b$, Precedence is an index-dependent relation such that each tick of the clock a has to precede (tick earlier than) the tick of the clock b with the same index. Precedence can be strict or not.

The idea of TRAP emerged from CCSL and relies on the same basic principle: a property expression specifies constraints on clock ticks and these constraints can be translated into a set of parallel automata operating on the clock flow. Each individual automaton possibly makes a transition on the occurrence of some clock tick. Eventually either it reaches a satisfying final state, or a property violation is detected.

Unlike CCSL, TRAP does not deal with infinite traces. A simulation session generates a finite trace and it is intended that all properties expressed should be verified within one session. When the simulation terminates, each property automaton that is not in the satisfying (Accepting) state reports a violation at the end of the analysis.

In TRAP, the property specification language, named TPSL, is not limited to only CCSL operators. Similarly to CCSL, TPSL defines properties that can be compiled into automata but these automata may capture more than pure CCSL operations or relations. Augmenting CCSL is necessary to capture some properties such as:

- the difference between necessary and sufficient conditions;
- the required absence of some particular clock C between two specified clocks A and B;
- constraints on clocks cardinality, such as those necessary to validate typical properties of CPS related to jitter, throughput, burstiness and latency, or simply buffer sizes.

Other extensions, like MoCCML [26] were proposed in a bid to extend the expressiveness or ease the ways properties can be expressed. Here, we are looking for a balance between the accessibility to a wide range of engineers and the expressiveness.

There are several solutions proposed to encode CCSL constraints into other synchronous languages, like Esterel [27] or Signal [28]. The goal here is pretty different since we keep the systemC separate and we use our DSLs to annotate the types in a way such that we have an automatic instrumentation of the SystemC code for producing trace streams of ticks instead of raw traces.

The implementation of observers is also very different from the one in [27] since the automata generated by TRAP are not regular finite state automata. Therefore, TRAP does not use the software library from the CCSL toolsuite TimeSquare [29], it relies on the notion of *symbolic finite state transducers* (SFTs) [30, 31], which are an extension of classical automata by allowing transitions to be labeled with

arbitrary formulas in a specified theory, in our case boolean expressions over tagged clock ticks (clock ticks tagged with time stamps).

The basic alphabet for the transducers is the clock alphabet, but the transducers may include predicates that operate over a finite set of clock ticks. When taking transitions, the transducers may add information to the *history* of the automaton. Each automaton carries in its history a summary of what happened during the evaluation of the property. It maintains data such as the timestamps and the counters of the first and last ticks of each clock involved in the automaton, or the absence thereof. The property specification language has been designed precisely such that maintaining the history is sufficient to evaluate the transducer predicates and no other data is required. The history has enough information to guarantee the decidability of the predicates translated from the property clauses. The history is reset to void each time an automaton enters its initial state.

In summary, although TRAP is based on the CCSL notion of logical clock, the generated automata are SFT's associated with a local history such that the predicates can be evaluated when taking transitions, augmenting the CCSL automata.

TRAP properties are verified over a finite execution time of the system producing some trace. There are no specific constraints on simulation sessions and the trace length but it is explicitly finite. The idea is that all properties in TPSL should be verified at the end of the simulation session. In other words, if a property checked would correspond to a temporal logic expression with an 'eventually' clause, the eventuality must have occured in the trace analysis, otherwise it is considered as a property violation. If such a propery is violated during a simulation session, it might be because the simulation session has been too short. The engineers may have to investigate such properties and define simulation conditions to avoid such situation, which occur unfrequently in our experience.

2.3 The SimSoC Framework

The work described here is based and integrated into SimSoC [32], an open-source virtual prototyping framework developed at Inria [33]. The code generated by the tools described below is compiled and linked with the SimSoC Instruction Set Simulator to generate traces. Although our work is based on SimSoC, it is independent and can be retargeted to other systems. A more extensive discussion on virtual prototyping issues is available in [34].

3 Runtime Verification

3.1 Global Architecture

The global software architecture of our approach is shown in Figure 1. It is decomposed into four independent steps. The

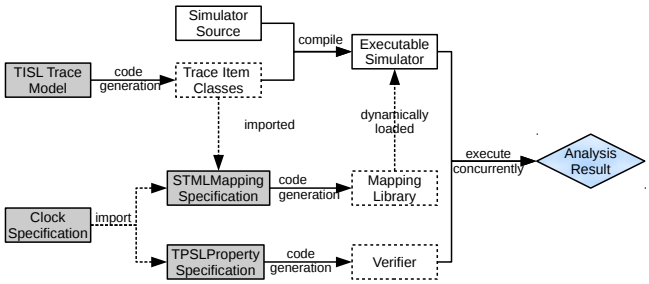

Fig. 1 Global architecture

first step is the trace generation, based on a trace meta model so that all traces are conforming to some well defined model. We use our DSL called TISL to specify the data types that should be instrumented. From this description we generate some SystemC code that is injected in the SimSoC model. The second step is a model transformation process, using a DSL called STML, in order to map the actual simulation binary trace data into an abstract trace of logical clocks. The third step is the property specification by means of another DSL called TPSL, an English-like pattern-oriented specification language. The final step is the property verification over the mapped trace, using code generated by the property specification compiler.

The individual steps, described in more details in the following sections, are motivated and structured as follows:

- We initially place a light constraint on the simulator: The simulator can generate arbitrary binary data into what is called the *raw trace* as long as each data item dumped into the trace can be modeled as a structure that can be described using the meta-model defined and explained in section 3.2. Because the trace corresponds to such a model, it can be parsed and understood for extraction and transformation. Another constraint on the simulator implementation is that it must comply with the trace interface used to produce trace items.
- The runtime verification is based on the concept of logical clock and the verification process eventually operates on logical clocks. Thanks to the meta-model, a mapping process can be defined, which maps raw

binary data into logical clock ticks. For example the fact that the variable value of a sensor has changed in some particular way can be transformed into a specific clock tick.

- This mapping of raw data is achieved using a DSL called STML (Simulation Trace Mapping Language), defined using the Eclipse Modeling Framework. Engineers can take advantage of STML to filter out useless data from the trace and significantly reduce the trace file size, or map complex data structures into clock ticks. The STML compiler generates the mapping code as a dynamically loadable module implementing the trace interface.
- After the raw trace has been mapped to a trace of logical clocks, it can be analyzed to verify the required properties, expressed in another DSL named TPSL (Trace Property Specification Language). TPSL has been designed to be a simple language usable by engineers that do not require knowledge of temporal logics or logical clock algebra. It has English keywords and support mathematical expressions using syntax similar to the C programming language. A TPSL module consists of an arbitrary number of properties. The TPSL compiler compiles each module into a set of parallel SFTs corresponding to the properties.

The resulting automata are applied to the mapped trace stream and result into either a PASS or FAIL status. In case of FAIL, an explanation is provided upon the failed expression in TPSL. Since the automata are parallel and no backtrack is ever operated, the verifier performance is linear with the trace stream size.

An important notion in CCSL is the notion of simultaneity. However, in a CPS, the requirements on simultaneity may be that some events occur "simultaneously" considered as within the same millisecond, whereas the system clock is expressed in nanoseconds. Successive clock ticks within the same millisecond must then be logically considered as simultaneous with regards to the property being verified. Since logical clocks represent observable events with a different scale, several successive trace items may also be mapped into a single logical clock.

In TRAP, the simulator must provide a timestamp for all data item produced, expressed in some unique real time clock, typically the hardware clock resolution. For the verification of properties, another real time clock is used, the *trace clock*. The mapping transformer is parameterized by the trace clock frequency (which must be a divider of the real clock frequency). The trace emitter finally emits logical clock ticks annotated with tags related to the trace clock. Thus, trace items that may have a different timestamp relative to the system real time clock may translate into simultaneous clock ticks as observed by the trace clock. The trace clock frequency is chosen by the users depending on their use cases.

3.2 Trace Emission

The role of the trace emitting module is to fix the representation of the binary data that is emitted by the simulator. To avoid recompiling and rebuilding the simulator when checking different properties, the trace generation is done inside a dynamically loaded module that is provided as a parameter when the simulation starts. The simulation must call this module using the provided interface. A key element of this interface is the `trace_emit()` function, an abstract polymorphic function that the simulation engineers must call from their models.

The interpretation of the raw trace data is defined by a UML model. This model must comply with a meta-model that is essentially the following: each data item in the trace must carry a timestamp and belong to some class. Each class has a name and an arbitrary number of attributes. Each attribute has a name and a data type, which must be a basic value type. Pointers and references are not supported. Embedded structures are not supported either at the moment, except fixed size arrays. These data items can represent anything, sensor values, states, transitions, events, transactions, no specific requirement is made.

The TISL (Trace Item Specification Language) DSL makes it possible to describe arbitrary trace model that comply with this meta model. A TISL trace model must be provided by the engineers constructing the simulator (using either an independent Bison tool or the Eclipse Modeling Framework) who can then output any piece of information described in the model. From this model, it is possible to derive trace generation code that captures and generates the simulation data into a well understood format, similarly to XML technology. A TISL specification allow the users to define which binary data is emitted as a sequence of instances of well defined classes. TISL supports an inheritance mechanism whereby an item class may extend a parent class, using a syntax similar to Java. The TISL compiler generates C++ code linked with some hand-written code that implements the trace_emit() interface.

Figure 2 shows a fragment of the TISL code used in our

```
type Byte "int8_t"
type Bool "bool"
type Address "uint64_t"
type Size "size_t"
type Target "uint_64_t"

enum FifoStatus {Some, Empty, Full}
enum Operation {Read, Write}
enum UART_Out_Signal {
  FifoEmpty, FifoFull, DataAvailable, Ready,
  Timeout, Error
}
item UART {
  Byte lsr;
  Byte msr;
  FifoStatus fifo;
}
item UART_Control {
  Operation operation ;
  Byte register;
}
item UART_Signal extends UART {
  UART_Out_Signal signal;
  Bool value;
}
item MPIC_Request {
  Byte line; //interrupt line number
}
item MPIC_Raised {
  Byte line; //interrupt line number
  Byte core; //destination core
  Byte priority; // interrupt priority
}
item Transaction {
  Operation cmd;
  Target target;
  Address addr;
  Size size;
}
```

Fig. 2 TISL example

experiments. They illustrate the approach using a virtual prototype built with SimSoC framework from Inria coded in SystemC that includes a Power architecture instruction set simulator. The virtual prototype is that of a multi-core Power platform. Each core has its own MMU (Memory Management Unit), and is connected to various peripherals, including a OpenPIC (MPIC) multi-core interrupt controller and a transmit (UART) controller with a FIFO queue. The interconnect is an abstract TLM bus. The TLM transactions have an operation, a target device, and a transaction address, for some data size. The trace model excerpt shows different trace items emitted by the UART hardware, all deriving from the parent UART class, which indicates the line status, the modem status and whether the fifo is Empty or Full. Other UART items derive from the parent and output additional state information, for example when the UART emits a signal. Similarly the interrupt controller MPIC emits some trace items when state changes. As the trace generator code is itself generated from the model, it can arrange the layout of the trace data as most convenient for later processing, while maintaining its integrity.

3.3 Trace Mapping and Filtering

From the above item definitions and event/clock definitions, there must be a specification defining how to convert trace items into clock ticks, i.e. a mapping language. The next step in the process consists in mapping the trace items to logical clocks, using another DSL, named STML (Simulation Trace Mapping Language), that builds the bridge between trace items and clocks. It defines how the trace items are mapped into clocks using pattern matching. This step is inspired by Open Trace Format 2 [19].

A STML module takes as input two parameters, on one hand the trace model described above, on the other hand a set of clock definitions. It then contains instructions that specify the mapping of the trace data item into either nothing (the data is eliminated from the trace) or into one or more clock ticks. The STML language also helps reducing the trace file size by either filtering out irrelevant data with regards to the properties to be verified or by combining complex binary data into a single clock tick. Another feature reducing the trace and facilitating property verification is the ability to emit clock ticks only when a value changes in some way specified with an arithmetic expression, eliminating the need to store repeatedly the same value over and over.

As mentioned in the introduction, there are properties that should hold only during specific phases of the running system. TRAP uses to that end the notion of *Scope*. A scope defines an arbitrary period of time and properties may be attached to scopes. TRAP is supporting overlapping scopes, not necessarily nested. At any time the system is within a list of scopes and a property may have to be verified only within such a list. STML thus also provides a mechanism to enter and leave a scope. There are no constraints on scope definition. Entering or leaving a scope can be indicated either by hardware state change or software state change.

An STML specification module consists of a header importing the TISL item and the clock specifications (the latter one also defines the scopes), then possibly introduce

global variable definitions, then a set of rules, each one introduced by the keyword `for` followed by an item class name defined in TISL. For each final item (i.e. not inherited by other items) in the trace model, there should be a corresponding mapping rule defined for it in the mapping specification, so that the mapping is guaranteed to be fully defined. By default, if a mapping is unspecified, all trace items of that class are simply ignored. Each rule describes how the trace items of this class translate (or not) into some logical clock occurrences, using a syntax close to the C language for expressions.

Each rule defines a pattern expression and must specify both what happens if the pattern matches and if it does not match. Each pattern maps the input items into one of the three possibilities:

- simply ignore the pattern because it is irrelevant to the properties considered in this analysis. This is notified by keyword `ignore` meaning no output clock ticks.
- enter or leave a scope with keyword `enter` or `leave`.
- emit a list of (simultaneous) clock ticks introduced by keyword `emit`.

Emitting a logical clock tick can be conditional, based on some variable value. For example, some interrupt related clock can be emitted only when some particular bits of the hardware interrupt registers are set.

To further reduce trace size and map related successive data items in the trace, STML allows the pattern for trace items to be also conditional to the occurrence of subsequent trace data. A pattern may specify subsequent trace items to reduce several successive data items into one logical clock if they meet some condition, for example emtting a signal from the UART causes a state change in the interrupt controller, creating a pending situation. A question then is: how long can possibly be that sequence? STML does consider only the simultaneous events (relatively to the trace clock) to map them into a single logical clock event. A pattern specifying a sequence of events is matched only if that pattern occurs in the same trace clock period.

To that end the mapping algorithm constructs a queue of the trace data items ordered by their timestamps in the simulation clock until the next period of the trace clock (the queue size increases as the trace clock frequency decreases). It then considers the earliest data and checks the STML rule for that item. If that rule contains a conditional path based on the eventuality of a later event, that event is searched in the queue. When a pattern is matched, all of the matching events are removed from the queue. Since STML requires that the rules always specify what to do in both cases (matching or not) this is always decidable. Hence, any trace data in STML is considered only once. The same data item cannot participate twice or more into generating a clock tick. However it may be that simultaneous trace data (with respect to the trace clock) generate simultaneous logical clock ticks.

In summary, the STML language, given the input trace model, makes it possible to:

- specify a conditional action. The condition must bear on the values of the attribute of items, or on the global variables. Each branch can be a basic action, or another nested conditional action.
- check for a sequence of items that finally construct a pattern. This sequence of items is specified using a construction similar to a `switch` instruction in C++ programming language, introduced by keyword `when` followed by candidate `case`, with a mandatory `else` clause in case the sequences of items specified do not match. This construction builds a pattern tree used by the pattern matching algorithm.
- assign an attribute value of an item to a global variable. Indeed in many cases the trace items contain state information that is duplicated over and over again in the trace, whereas the verification is only interested in state changes. Global variables in the language make it possible to emit clock ticks when some particular global values has changed in a particular way, with a conditional that makes it possible to quantify that change. For example, it is possible to express the case “emit a HEAT clock tick if the temperature sensor has increased by more than 10% compared to last time considered”.

By defining such mapping rules for all possible trace items, any trace stream can be mapped into clock ticks to generate an abstract trace. Since the mapping process may ignore some data, or reduce repeated state to state changes only, and reduce multiple items into a single clock tick, the size of trace streams can be remarkably reduced when studying a given set of properties.

The STML compiler turns the mapping rules into a dynamic linking library, loaded by the simulator. This dynamic library is called by the `trace_emit()` interface and implements the specified mapping, generating a clock stream. Therefore one can dynamically generate as many traces as desired from the same simulator, without rebuilding the virtual prototype (which may not be an easy and quick task). Users may also, using static analysis, store

the raw trace as a huge file, using no filtering, but the library can be reused to transform that huge file into as many clock mappings as desired.

The logical clock stream can then be analyzed with automata resulting from property specification compiler. In STML example on Fig. 3, only data items related to sending from and receiving to the FIFO are mapped in order to analyze the FIFO handling behavior. Only the items related to the FIFO are mapped into clock ticks. All other trace data items are simply thrown away. Regarding interrupts, the trace items emitted by the interrupt controller translate into different clock ticks depending on the successive trace items as the interrupt controller is tracing both whether it receives a signal and whether the corresponding interrupt is enabled. In the latter case, the interrupt may remain pending (e.g. because there is a higher priority interrupt) or it is raised to some core. These different situations map to different clock ticks.

```
for Transaction t { ignore }
for Signal s {
  s.signal == FifoIsEmpty ?
     emit FifoEmpty
     : s.signal == FifoIsFull ?
        emit FifoFull : ignore
}
for Send s { emit Sent }
for Receive r { emit Received }
for Control c {
  (c.operation==WRITE && c.target == Fifo
  && c.value == FLUSH ) ?
     emit Flush : ignore
}

for MPIC_Request r {
  (r.line == UART_LINE) ?
     when MPIC_Pending p:
     either MPIC_Raised s :
        emit Pending; emit Raised
        else emit Pending
     else ignore
   : ignore
}
```

Fig. 3 STML example

Any raw trace can be reduced to a stream of logical clock ticks with STML. Since the STML compiler must handle all possible classes specified in the TISL trace model, the user must specify for each TISL class how it translates into clock ticks (or whether it is ignored). Users may have to specify a fairly large number of clocks to distinguish specific cases. When specifying TPSL properties as shown in next section, it is possible to either specify specific properties for specific clocks or to group them using the union operator. After mapping, the engineers are able to focus on the clocks and scopes that provide a higher abstraction vocabulary for reasoning on properties.

It is important that this transformation does not alter the semantics. We do not give a full proof of that but rather an intuition about what this is true. A clock ci ($\langle C_i, \preceq_i \rangle$) are defined as a set of ticks C_i with a total order ($\preceq_i$). Each clock defines a total order on its ticks and a set of clocks defines a partial order on the union of their ticks. Some relations force some ticks to coincide (be merged) while others just impose an order on ticks of different clocks. We have two assumptions, first we treat the events in the order of their timestamp, a total order given by the machine. Second, we only produce ticks based on events that have the same timestamp. With respect to a logical clock, all these events are coincident, it is just fair that they are transformed into a single tick, i.e., an atomic execution that is totally ordered with respected to the other ticks of the same clock. So if two events $e1$ and $e2$ of the initial traces are such $time(e1) \leqslant time(e2)$ then $f(\{\ldots, e1, \ldots\}) \preceq f(\{\ldots, e2, \ldots\})$, when $time : Event \rightarrow \mathbb{N}$ is the function that extracts a timestamp from an event, $\leqslant$ is the natural order on integers, $f : 2^{Event} \rightarrow Clock$ is the function that transforms a set of events matching a given pattern into a clock and $\preceq$ is the partial order on clocks defined on $\bigcup_i(C_i)$ as the union of the total order on clocks $\bigcup_i(\preceq_i)$. The operator *ignore* will remove some events from the trace without changing the order of others. This is not a problem either since we are only allowed to express properties on events that are not ignored. The next subsection describes the language used to express those properties.

3.4 The Property Specification Language

Following the same reasoning as Balarin et al. [6] or Drechler et al. [35], we advocate for a specification language close to the domain and easy to compute rather than as expressive as temporal logics but failing to get a wide adoption among engineers. Another domain-specific language, named TPSL, is dedicated to expressing properties on simulation traces. TPSL[1)] also uses the Eclipse Modeling Framework for easy integration in development tools so that engineers can express properties that should be verified on traces. This language allows to import the list of logical clocks and scopes to be generated by the mapping language. This language captures the properties that cover a large part of the CPS engineering applications:

- causality: the fact that when some event occurs or some

[1)] not to be confused with PSL [36]

state is reached, there should be a consequence in the future. Causality can be altered by presence or absence of some other event.
- delay: the fact that two events should not be spaced by more or less than a given (time) value.
- counting occurrences of events and comparing counts.
- specific well known properties such as jitter, throughput, latency and drift.

A TPSL module must first import the clock identifiers and the constant values that are used in subsequent properties. These declared clocks are called the *primary clocks*. Users may then define additional new clocks from the primary ones using one of the following operators:

- **X = A or B** to define X as the CCSL union of the two clocks A and B.
- **X = A and B** to define X as the CCSL intersection.
- **X = A by N** to define X as the CCSL subclock of A corresponding to frequency divider N.

After the clock definitions, users may define properties related to these clocks. The language has a few English keywords to construct properties. The property language considers only two relations from CCSL, the alternation (CCSL notation $\sim$) of two clocks, that ticks alternatively, and the precedence of two clocks (CCSL notation $\prec$ or $\preccurlyeq$).

The properties follow a generic pattern, which is:

```
Scope Clock_Expr Relation Clock Constraints
```

The scope can be either **always** if the property must hold at any time (the default value), or **never** if something must never happen. Most of the time, a Clock Expression results into a *hidden clock*, that is, a clock internally generated by the system, resulting from evaluating the clock expression. Thus, a property is always waiting for a specific clock, either from a primary clock or a hidden one. The hidden clocks are generated by automata that directly implement CCSL operators, or from more complex transducers. In particular TPSL allows some kind of regular expression pattern matching on clock ticks, using two operators denoted < and <>. $A < B$ is in fact equivalent to the regular expression A[!B]*B, that is at least one tick of A is followed by at least one tick of B, but there can be any other intervening clock between A and B. Similarly $A <> B$ is matched whenever A is followed by B or B is followed by A. These pattern matching expressions must not be confused with the precedence relation $\prec$ of CCSL. The relation following a clock expression is either **causes** or **alternates** to mean that the two clocks must constantly alternate.

The causality relation is refined in TPSL because it is often necessary to distinguish between necessary and sufficient conditions. For example, in a train scenario, it is sufficient that the alarm is raised to stop the train, but the train may stop for other reasons. On the contrary, in a embedded software context, whenever an interrupt is raised, there should be a reason for this interrupt, otherwise it is spurious. But a pending interrupt may be disabled and never occur. TPSL uses the keywords 'causes' to express the sufficient condition and the condition becomes necessary by adding the '!' symbol. Thus in TPSL, `Alarm` **`causes`** `TrainStop` and `Signal` **`causes!`** `Interrupt`.

Another distinction in causality is uniqueness. It may be the case that multiple ticks of some clock should result in one single consequence (several reasons can cause the alarm but the train stops only once) or conversely that each action will result into another action (each message sent should be received). The keyword **each** is used to specify the latter. This is in fact an additional constraint on the CCSL precedence relation. In CCSL, if A $\prec$B, the clock ticks of B corresponding to those of A (by index value) may be delayed infinitely. TPSL requires that those clock ticks are present in the trace before the end of the simulation session.

In addition to the relationships, there may be additional constraints on the property and we distinguish three of them:

- additional conditions that must be satisfied. These conditions may be related to the respective timing of the clock ticks, or the count of tick occurrences. The keyword **satisfies** expresses such conditions. It means that for the property to be verified, not only the clock ticks must obey the rules but there are additional constraints on these clocks, typically a time delay between some ticks;
- suspending condition for the causality, introduced by keyword **unless**, to express that a property has to be verified unless something happened, for example a Cancel order (e.g. `Signal` **`causes`** `Interrupt` **`unless`** `Disabled`). The semantics is that the property is void if the unless clause did occur between the first and the last relevant clock ticks in the trace. The unless clause makes it possible to deal with absence in a number of cases, using a property template of the form `Something` **`causes`** `Error` **`unless`** `OK`;
- activation condition for the property, introduced by keyword **if**. The semantics is that the property holds

only if the condition is true. This is different from unless clause as the predicate is not a clock expression but a Boolean expression either relative to event occurrences or with respect to a time stamp. For example, writing into the FIFO may cause the FIFO to be full only if the previous items were not sent already.

- Causality properties specifying **each** for some clock C have a slightly different behavior. No **if** clause can be specified then as it would be contradictory; and the unless clause, if present, only applies to the most recent occurrence of C, but it does not void the causality of previous occurrences.

In addition, TPSL has specific constructs.

The property `jitter(clock, start, period, margin)` is defined to ensure the deviation from true periodicity of a presumably periodic signal within some margin. After timestamp *start*, all ticks of *clock* identified by their occurrence counter i (starting at 1 for the first tick) must satify the relation $start + (i * period) - margin <= timestamp(i) <= start + (i * period) + margin$ where period and margin are integer values in units of the reference clock. The automaton history does not need to maintain previous occurrences of the target clock.

The property `throughput(clock, duration, count)` expresses that for all ticks of *clock* identified by their counter i (starting at 1) with $i > count > 1$ the relation $timestamp(i) - timestamp(i + 1 - count) <= duration$ must hold. The first *count* ticks are added to the automaton history. After that, if the property is satisfied, the earliest tick is removed from history and the newest tick is added.

The property `burstiness(clock, duration, count)` expresses that for all ticks of *clock* identied by their counter i (starting at 1) with $i > count > 1$, the relation $timestamp(i) - timestamp(i - count) >= duration$. Similar to throughput, the automaton history contains the *count* most recent ticks.

4 Implementation and Tests

4.1 Implementation

STML is a DSL described in Eclipse Modeling Framework [37] using the Xtext tool and an abstract syntax model also using Ecore. The STML compiler is coded in Java. It generates the C++ code of the dynamic mapping library. This dynamic library is loaded by the simulator that generate trace streams. One can change the STML rules to generate different clocks to check different properties without recompiling the simulator. The SimSoc simulator has two options to either generate the trace stream into a file or dynamically send it to a verifier process for run time analysis.

TPSL is also described as a DSL in the Eclipse EMF framework. The TPSL compiler generates code for the Verifier. The verifier consists of two main units, the dispatcher and the set of parallel automata. The dispatcher dispatches basic clock ticks to the destination parallel automata. Since the TPSL compiler knows which clock operators and which automata depends on which clock ticks, it can selectively dispatch the basic clock input to their destination. A clock tick may be dispatched to multiple automata as the rules are verified in parallel.

By construction of TPSL, the automata all follow some particular template. The TPSL compiler has a library of such templates that it can instantiate with the particular clock expressions and predicates from the source code. This can be implemented using C++ template facility combined with a functional style since C++ standard now supports lambda expressions. Figure 4 provides the diagram of the automaton for the template

⟨*ClockExpr*⟩ **causes** ⟨*ExpectedClock*⟩
if ⟨*Condition*⟩
unless ⟨*Constraint*⟩

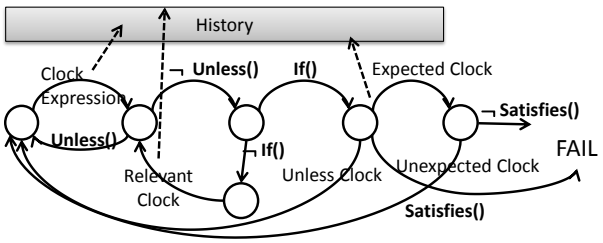

Fig. 4 Automaton template with if and unless clauses

Each template automaton has a specific behavior, but they all follow the same generic pattern: from the initial state (when it is active), each transducer reaches a checking state when the clock expression has been encountered. The clock expression is actually a dependent automaton, also generated from a template. The transition is taken in the property automaton when the dependent clock expression automaton terminates. Based on the predicates result, the transducer will either return to the initial state, or stay in the same state,

or move to the expecting state. It then expects some clock tick to occur to meet the required causality. If it does, it will transition to the satisfying state, and then, depending on the required condition to be satisfied, which bears on the history, it returns to its initial state or fails. While in the expecting state, other ticks may occur, for example a tick that would make true the unless clause, or ticks that modify the count in the history. Such clock ticks are told to be *Relevant Clocks*. An automaton may take a transition when some such relevant tick occurs. Again, the TPSL compiler knows what these relevant ticks are and can generate code that dispatches only such relevant ticks to each automaton, and generate appropriate tests in the automata to take the transition.

The **each** automaton template is particular. As shown on Fig. 5, it is a potentially infinite automaton that waits for *N* expected clocks after *N* causing clocks. It will return to initial state only when the counts of occurrences are equal.

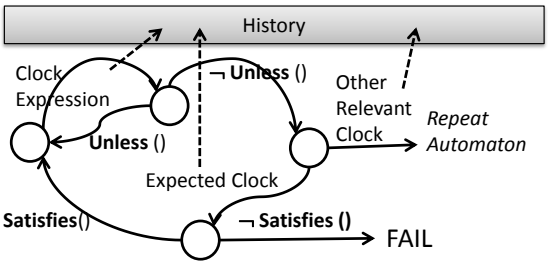

Fig. 5 Automaton template for each clause

The verifier takes as input a trace stream and outputs either a success message or one or more failures in the properties verification. Some part of the verifier is manually coded, essentially a library of base functions, the automaton templates, and the skeleton of the main program. This predefined code is linked with the code generated by the TPSL compiler to form a program that can read trace streams. It works essentially as follows: each line of a trace stream has a time stamp (in the trace clock) followed by all clock ticks that occured on that timestamp.

The verifier first calls a setup function that creates all of the parallel automata generated by the TPSL compiler and it activates those that are in the **always** or **never** scope. It then repeatedly reads the trace stream line by line and calls the dispatcher, which does dispatch the clock ticks to each individual active automata.

During this process some automaton may become active or inactive, in particular when a scope is entered or exited, and some may report failure. At the end of the trace stream, the automata are checked again to know if they have reached their final state. For any property, if the end of the trace is reached while a transducer is still in expecting state, it is a failure.

In addition to the verifier, TRAP is providing a debugger in order to investigate property violations. The debugger is invoked by providing a `-debug` argument to the verifier program. The debugger has a command line user interface and is responsible for:

- Calling functions provided by the mapper to parse the trace and get clock ticks one by one;
- Calling functions to enter each tick obtained from the previous step to the automata. If any error happens, error signal emitted by the is caught by the debugger and appropriate action will be performed.
- Accepting user input and passing arguments to corresponding command operator, which will then give information or change some state according to the user command.
- Managing the breakpoints. A user can set breakpoints so that the debugger will pause and let user input command again.

4.2 Tests

We have instrumented the SimSoC PowerPC Instruction Set Simulator and several simulation modules to generate traces according to our model. The main experiment simulates a multi-core PowerPC processor connected to various peripherals, including a OpenPIC multi-core interrupt controller and a transmit controller with a FIFO queue, controlled by a real time monitor that has drivers monitoring the peripherals. The hardware FIFO queue receives items asynchronously from the software but it re-sends these items synchronously. The hardware should (i) emit a $FifoEmpty$ signal whenever the queue is empty (including at initialization phase), a $FifoFull$ signal when it is full, (ii) send the data items with limited jittering (iii) have a reasonable latency. On the other hand, the software should (i) enable interrupts and service them, and (ii) it should not put data into the queue when it is full.

A $FifoEmpty$ signal should be received by the interrupt controller. In our application this interrupt should not be disabled and it should become pending. However the interrupt controller may choose which core should receive that interrupt and only one of the cores should receive it.

The embedded software is a test program that emits data to the outside world, using the drivers and receiving interrupts. The software must enable interrupts, handle and

acknowledge the interrupts with the hardware and service them, within a time constraint. When the interrupt is serviced both the interrupt controller and the UART must return into normal state. On a multi-core configuration like the one considered, this induces a cascade of acknowledgments and signals. The full simulation trace contains data regarding hardware states, transactions between the cores and the controllers and events happening consequently to software actions.

The TPSL code regarding this example (see Fig. 6):

```
clock Put Send FifoFull FifoEmpty;
clock Signal Interrupt Pending ACK;
constant int FifoSize = 16;

//the hardware must not send when fifo empty
never Send between FifoEmpty Put;

// the software must not put when fifo full
never Put between FifoFull Send;

// each put must be balanced by one send
Put each causes Send
Put causes FifoFull
      if count Put - Send = FifoSize;
Send causes FifoEmpty
     if count Send - Put = FifoSize;
FifoEmpty causes Signal;

// after Signal has been detected as Pending
// there must be an interrupt interrupt handled
// by service routine that must ACK
Signal < Pending causes! Interrupt
causes! ACK
within ACK - Signal < INTERRUPT_HANDLING_DELAY
```

Fig. 6 TPSL code for testing the interrupt controller

In this case, the trace reduction is significant, as we are only verifying properties related to the FIFO queue and consequently many other hardware states are simply ignored. The trace file emitted by the test on SimSoC for a duration of 27 seconds is 21.162 Megabytes, the size of the logical clock trace is only 1.737 Megabytes, a reduction of over 90%. The trace is analyzed by our tool in less than 1 second. This is achieved through the STML code given in Figure 7.

Thanks to this framework, problems were discovered in one of the test cases. The UART controller raises interrupt signals to the interrupt controller with a level sensitive signal. When a UART interrupt is enabled and pending, the interrupt controller dispatches the interrupt to one of the available cores. The interrupt handler software then must acknowledge the interrupt, first with the UART, next with the interrupt controller. When the software acknowledges the interrupt with the UART, the signal goes down.

```
for UART_Signal u {
(u.signal=up) ? emit UARTSignalUp
: emit UARTSignalDown;
}
for MPIC_Request i {
(i.line == UART_LINE) ?
(i.signal==up) ? emit UARTPending
: emit UARTNotPending
: ignore;
}
```

Fig. 7 STML code for our UART

However, another UART interrupt that is pending (but waiting) may raise the signal again immediately, and this is typically done before the software has acknowledged the previous interrupt with the interrupt controller. Interrupts in the virtual prototype were not handled properly, for two reasons. First the SystemC model for UART was implemented as a single process. When the signal was going down and up, is was implemented as two consecutive non blocking TLM transactions, without waking up the interrupt controller model. Therefore the signal change was unnoticed. This defect was uncovered by the simple specification shown in Figure 8. As the clocks are emitted by the two different SystemC models, the second clock was simply not emitted.

```
UARTSignalUp alternates UARTSignalDown
UARTPending alternates UARTNotPending
UARTSignalUp each causes! UARTPending
UARTSignalDown each causes! UARTNotPending
```

Fig. 8 TPSL code for our UART

This defect was fixed by modifying the UART hardware model, which uncovered the second defect. This second defect was that, when the UART signal would go up again before the acknowledgment of the first interrupt by the software handler, the state change was managed incorrectly in the model. Although an interrupt is raised, a second interrupt of the same type can be pending.

5 Conclusion

We have demonstrated the feasibility of runtime verification of system properties for investigating system failure from simulation trace analysis, by abstracting raw binary simulation trace files into logical clock trace files. Thanks to a DSL that we have defined based on a simple yet powerful trace meta model, the huge binary trace files can be reduced

and mapped to much smaller abstract traces relating occurrences of logical clock ticks. A language has been defined to capture temporal properties, compiled into parallel automata executed by a runtime verifier.

Our goal is not to formally offer more expressive power than any variant of temporal logic. Our tool is meant to be a convenience tool for engineers that have limited knowledge of formal methods but still can express temporal and causal properties in a easy-to-use language that is supported by a user-friendly environment, namely the Eclipse platform.

Another advantage of our technique is that it does not require to recompile the simulator to change the mapping and verify different types of properties. Different mappings can be achieved and different properties verified from a single trace. This provides a mechanism for engineers to explore several paths for investigating a failure cause, either recording a single trace file or replaying the same simulation session with different clock mappings and different clock properties.

The SimSoC implementation has been modified so that a new argument can specify the trace mapping module to be loaded. In such case, the trace stream is generated such that it can be pipelined with the verifier. The verification can occur at runtime, concurrently with the simulation (possibly using at least two of the cores on a multi-core platform, introducing no penalty on the runtime verification), somehow like an extension of an assertion verifier, except that the assertions can be modified and checked without recompiling the simulator. In addition, our system can also work as a static analyzer for trace data stored into files.

In the current implementation, the mapping language translates binary data into genuine clock ticks. We are considering in the future to associate attributes with the clock ticks, then the predicates in the transducers could also evaluate Boolean expressions over such attributes.

Acknowledgements This work was supported by the Sino-European LIAMA Laboratory and by the INRIA Sophia Antipolis Research Center.

References

1. Dahan A, Geist D, Gluhovsky L, Pidan D, Shapir G, Wolfsthal Y, Benalycherif L, Kamidem R, Lahbib Y. Combining system level modeling with assertion based verification. In: Sixth Int. Symp. on quality electronic design (isqed'05). March 2005, 310–315
2. Foster H. Assertion-based verification: Industry myths to realities. In: Computer Aided Verification. 2008, 5–10
3. Krahl D. Debugging simulation models. In: Proceedings of the Winter Simulation Conference, 2005. Dec 2005, 7 pp.–
4. Leucker M, Schallhart C. A brief account of runtime verification. The Journal of Logic and Algebraic Programming, 2009, 78(5): 293–303
5. Pnueli A. The Temporal Logic of Programs. In: Proceedings of the 18th IEEE Symposium on Foundations of Computer Science. 1977, 46–57
6. Balarin F, Burch J, Lavagno L, Watanabe Y, Passerone R, Sangiovanni-Vincentelli A. Constraints specification at higher levels of abstraction. 2012 IEEE International High Level Design Validation and Test Workshop (HLDVT), 2001, 0: 129
7. Chen X, Hsieh H, Balarin F, Watanabe Y. Automatic trace analysis for logic of constraints. In: 40th Design Automation Conference. 2003, 460–465
8. Hong W, Viehl A, Bannow N, Kerstan C, Post H, Bringmann O, Rosenstiel W. Cult: A unified framework for tracing and logging c-based designs. In: System, Software, SoC and Silicon Debug Conference (S4D), 2012. Sept 2012, 1–6
9. Tabakov D, Kamhi G, Vardi M Y, Singerman E. A temporal language for systemc. In: Formal Methods in Computer-Aided Design, 2008. FMCAD'08. 2008, 1–9
10. Khlif M, Shawky M, Tahan O. Co-simulation trace analysis (cosita) tool for vehicle electronic architecture diagnosability analysis. In: Intelligent Vehicles Symposium (IV), 2010 IEEE. 2010, 572–578
11. Bhargavan K, Gunter C, Kim M, Lee I, Obradovic D, Sokolsky O, Viswanathan M. Verisim: Formal Analysis of Network Simulations. IEEE Transactions on Software Engineering, 2002, 28(2): 129–145
12. Kim M, Viswanathan M, Ben-Abdallah H, Kannan S, Lee I, Sokolsky O. Formally Specified Monitoring of Temporal Properties. In: 11th Euromicro Conference on Real-Time Systems. 1999, 114–122
13. Dwyer M B, Avrunin G S, Corbett J C. Patterns in property specifications for finite-state verification. In: Int. Conf. on Software Engineering. 1999, 411–420
14. Konrad S, Cheng B H. Real-time specification patterns. In: 27th Int. Conf. on Software Engineering. 2005, 372–381
15. Some S, Dssouli R, Vaucher J. From scenarios to timed automata: Building specifications from users requirements. In: Asia Pacific Software Engineering Conference. 1995, 48–57
16. Tsai W T, Yu L, Zhu F, Paul R. Rapid embedded system testing using verification patterns. Software, IEEE, 2005, 22(4): 68–75
17. Tokarnia A M, Cruz E P. Scenario patterns and trace-based temporal verification of reactive embedded systems. In: Digital System Design (DSD), 2013 Euromicro Conference on. 2013, 734–741
18. Hegedüs Á, Ráth I, Varró D. Replaying execution trace models for dynamic modeling languages. Periodica Polytechnica. Electrical Engineering and Computer Science, 2012, 56(3): 71
19. Eschweiler D, Wagner M, Geimer M, Knüpfer A, Nagel W E, Wolf F. Open trace format 2: The next generation of scalable trace formats and support libraries. In: PARCO. 2011, 481–490
20. Hamou-Lhadj A, Lethbridge T C. A metamodel for the compact but lossless exchange of execution traces. Software & Systems Modeling, 2012, 11(1): 77–98

21. Mallet F, André C, Simone d R. CCSL: specifying clock constraints with UML/Marte. Innovations in Systems and Software Engineering, 2008, 4(3): 309–314
22. Lamport L. Time, clocks, and the ordering of events in a distributed system. Commun. ACM, 1978, 21(7): 558–565
23. Benveniste A, Caspi P, Edwards S A, Halbwachs N, Le Guernic P, Simone d R. The synchronous languages 12 years later. Proceedings of the IEEE, 2003, 91(1): 64–83
24. Gascon R, Mallet F, DeAntoni J. Logical time and temporal logics: Comparing UML MARTE/CCSL and PSL. In: Int. Symp. on Temporal Representation and Reasoning, TIME'11. September 12-14 2011, 141–148
25. André C. Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Research Report RR-6925, INRIA, 2009
26. Deantoni J, Diallo I P, Teodorov C, Champeau J, Combemale B. Towards a meta-language for the concurrency concern in DSLs. In: 2015 Design, Automation Test in Europe Conference Exhibition (DATE). March 2015, 313–316
27. André C, Mallet F. Specification and verification of time requirements with CCSL and esterel. In: Kirsch C M, Kandemir M T, eds, LCTES. June 2009, 167–176
28. Yu H, Talpin J, Besnard L, Gautier T, Marchand H, Guernic P L. Polychronous controller synthesis from MARTE CCSL timing specifications. In: Singh S, Jobstmann B, Kishinevsky M, Brandt J, eds, 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011. 2011, 21–30
29. DeAntoni J, Mallet F. Timesquare: Treat your models with logical time. In: 50th Int. Conf. on Objects, Models, Components, Patterns, TOOLS'12. 2012, 34–41
30. Noord v G, Gerdemann D. Finite state transducers with predicates and identities. Grammars, 2001, 4(3): 263–286
31. Veanes M, Hooimeijer P, Livshits B, Molnar D, Bjorner N. Symbolic finite state transducers: Algorithms and applications. SIGPLAN Not., 2012, 47(1): 137–150
32. Helmstetter C, Joloboff V. SimSoC: A SystemC TLM integrated ISS for full system simulation. In: IEEE Asia Pacific Conference on Circuits and Systems, APCCAS. November 2008. `http://formes.asia/cms/software/simsoc`
33. INRIA . SimSoC open source software. `http://gforge.inria.fr/projects/simsoc`
34. Joloboff V, Gerstlauer A. Virtual prototyping of embedded systems: Speed and accuracy tradeoffs. In: Nakajima S, Talpin J P, Toyoshima M, Yu H, eds, Cyber Physical System Design from an Architecture Analysis Viewpoint: Communications of NII Shonan Meetings, 1–31. Springer Singapore, Singapore, 2017
35. Drechsler R, Soeken M, Wille R. Formal specification level: Towards verification-driven design based on natural language processing. In: Forum on Specification and Design Languages, FDL'12. Sept 2012, 53–58
36. IEEE . Property Specification Language (PSL), 2010
37. Steinberg D, Budinsky F, Merks E, Paternostro M. EMF: eclipse modeling framework. Pearson Education, 2008

Daian Yue is a Master Student at East China Normal University. During the last year he has been working at INRIA Research Center in Rennes under the supervision of Vania Joloboff. He is working on the definition of simulation tools for the verification of systems mixing both hardware and software artefacts.

Vania Joloboff is a Research Director at Inria, the French National Research Institute in Information technology and control. He is the former European Director of LIAMA, the sino-european Laboratory of Informatics, Applied Mathematics and Automation in China, now leading a research group on the development of cyber physical systems at the Trustworthy Software Joint International Lab hosted at East China Normal University.

Prior to joining LIAMA, Dr. Joloboff was Chief Technology Officer at Silicomp Group, a French SME specialized in embedded systems design and emerging technologies, leading company innovation. He has also been Technical Director at the Open Software Foundation Research Institute.

Frederic Mallet is a Professor of Computer Science in Université Cote d Azur. He works on the definition of sound models and tools for the design and analysis of embedded systems and cyber-physical systems. He is a permanent member of the Kairos team, a joint team between Inria Sophia Antipolis research center and I3S Laboratory, a joint CNRS research unit.